Deckhouse Prom++ Documentation

Generated: July 06, 2026

Documentation

Concepts

Data model

Deckhouse Prom++ stores all data as time series: streams of timestamped values belonging to a single metric and a single set of labels. In addition to stored time series, Deckhouse Prom++ can generate derived time series as a result of query execution.

Metric names and labels

Each time series is uniquely identified by its metric name and optional key-value pairs called labels.

Metric names

- **Description**: Specifies the general characteristics of the system being measured (for example, `http_requests_total` — the total number of HTTP requests received).
- **Naming rules**:
  - Metric names can contain ASCII letters, numbers, underscores, and colons.
  - Must match the following regular expression: `[a-zA-Z_:][a-zA-Z0-9_:]*`.

> **Warning.** Colons are reserved for user-defined rules. They should not be used in exporters or direct instrumentation.

Labels

- **Description**: Allow the Deckhouse Prom++ data model to identify any combination of labels for a single metric. They define the specific dimensional implementation of that metric (for example, all HTTP requests that used the `POST` method for the `/api/tracks` handler). The query language allows you to filter and aggregate data based on these dimensions.
- **Naming rules**:
  - Labels can contain ASCII letters, numbers, and underscores. They must match the following regular expression: `[a-zA-Z_][a-zA-Z0-9_]*`.
  - Label names beginning with `__` (two underscores) are reserved for internal use.
  - Label values can contain any Unicode characters.
  - Labels with an empty value are considered equivalent to no label.
- **Changing labels**: Any change to a label value, including adding or removing labels, creates a new time series.

Samples

Samples form the actual data of time series. Each sample includes:

- **Values**: Floating point number (`float64`).
- **Timestamps**: Accuracy up to milliseconds.

> Starting with Deckhouse Prom++ v2.40, experimental support for native histograms has been added. Instead of a simple `float64` value, a sample can now represent a complete histogram.

Metric representation format

The following format is used to identify time series:

```
<metric name>{<label name>=<label value>, ...}
```

For example, a time series with the metric name `api_http_requests_total` and labels `method="POST"` and `handler="/messages"` can be written as follows:

```
api_http_requests_total{method="POST", handler="/messages"}
```

Service labels and metrics

In Deckhouse Prom++ terms, the endpoint from which metrics can be collected is called an instance—usually a single worker process.
A set of instances with the same purpose—for example, replicas of the same process used for scaling or fault tolerance—is called a job.

The following is an example of an API server job with four instance replicas:

```
- job: `api-server`
  - instance 1: `1.2.3.4:5670`
  - instance 2: `1.2.3.4:5671`
  - instance 3: `5.6.7.8:5670`
  - instance 4: `5.6.7.8:5671`
```

Automatically generated labels and time series

When Deckhouse Prom++ polls the target endpoint (target), it automatically adds several service labels to the collected time series to help identify the data source:

- `job` : Name of the job to which the target system belongs, according to the configuration.
- `instance` : Part of the target service URL in the `<host>:<port>` format from which the data was collected.

If any of these labels (`job` or `instance`) is already present in the source data, the behavior is determined by the `honor_labels` setting. For details, refer to the data collection (scraping) configuration section.

Each time data is collected (scraped) from an instance, Deckhouse Prom++ creates several additional time series:

- `up{job="<job-name>", instance="<instance-id>"}` : Equals to `1` if the instance is available (scrape was successful) and `0` if the poll failed.
- `scrape_duration_seconds{job="<job-name>", instance="<instance-id>"}` : Polling duration in seconds.
- `scrape_samples_post_metric_relabeling{job="<job-name>", instance="<instance-id>"}` : Number of metrics after applying metric relabeling rules.
- `scrape_samples_scraped{job="<job-name>", instance="<instance-id>"}` : Total number of metrics received from the target system for the poll.
- `scrape_series_added{job="<job-name>", instance="<instance-id>"}` : Approximate number of new time series added per poll.

The `up` time series is widely used to monitor instance availability.

If the `extra-scrape-metrics` flag is enabled, the following metrics are additionally available:

- `scrape_timeout_seconds{job="<job-name>", instance="<instance-id>"}` : Configured `scrape_timeout` value for the target system.
- `scrape_sample_limit{job="<job-name>", instance="<instance-id>"}` : Configured limit on the number of metrics (`sample_limit`) to poll. If no limit is specified, a value of `0` will be returned.
- `scrape_body_size_bytes{job="<job-name>", instance="<instance-id>"}` : Size of the last poll response (in bytes) if the poll was successful. If an error occurs due to exceeding the `body_size_limit` , `-1` will be returned; in other cases of unsuccessful polling, `0` will be returned.

Metric types

Counter

A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero upon restart. For example, you can use a counter to represent the number of requests served, tasks completed, or errors.

Do not use a counter to represent a value that can decrease. For example, do not use a counter for the number of currently running processes; use an indicator instead.

Gauge

A gauge is a metric that represents a single numerical value that can increase or decrease arbitrarily. Gauges are typically used to measure quantities such as temperature or current memory usage, as well as for “counts” that can increase and decrease, such as the number of concurrent requests.

Histogram

A histogram records individual measurements (such as request durations or response sizes) and distributes them across customizable ranges called buckets. It also stores the sum of all recorded values.

A histogram with the base name of the metric `<basename>` exposes several time series when collecting data:

- Cumulative counters for each bucket in the format `<basename>_bucket{le="<upper limit inclusive>"}`
- The sum of all recorded measurements — `<basename>_sum`
- The total number of recorded events — `<basename>_count` (equivalent to the value `<basename>_bucket{le="+Inf"}`, which includes all measurements)

To calculate percentiles based on histograms, you can use the `histogram_quantile()` function.

Histograms are also suitable for calculating the Apdex index. When working with buckets, remember that histogram values are cumulative.

Summary

Similar to histogram, summary records individual measurements (e.g., request durations or response sizes).

It stores the total number of events, the sum of all recorded values, and calculates customizable percentile values within a rolling time window.

A summary with the base metric name `<basename>` exposes several time series when collecting data:

- Percentile values φ, calculated on the fly, in the format `<basename>{quantile="<φ>"}`
- The sum of all fixed measurements — `<basename>_sum`
- The total number of recorded events — `<basename>_count`

Deckhouse Prom++ Server

Installation

WAL conversion before installation

Deckhouse Prom++ uses an alternative WAL (Write-Ahead Log) format, but remains fully compatible with historical blocks.
Since WAL contains the **last 1.5 blocks of data** (usually about **3 hours**), if you plan to use Deckhouse Prom++ as a replacement for Prometheus, you should convert WAL to prevent data loss.

See the migration guide for detailed conversion steps.

Precompiled binary files

1. Download the latest binary file considering the required architecture:
    - amd64
    - arm64
2. Unpack it:

    ```
    tar zxf prompp-binaries-<amd64|arm64>.tar.gz
    ```

    This will create a `prompp` folder with the `prompp` binary file and the `prometheus.yml` configuration file.
3. Run it as a Prometheus replacement:

    ```
    cd prompp
    ./prompp --config.file=prometheus.yml --storage.tsdb.path=data/
    ```

Docker

Deckhouse Prom++ is available as a Docker image in the following registries:

- `registry.deckhouse.io/prompp/prompp`
- `ghcr.io/deckhouse/prompp`

All available versions can be found on the Releases page.

To quickly start the container, run the following command:

```
docker run --name prompp -d -p 127.0.0.1:9090:9090 registry.deckhouse.io/prompp/
prompp:0.7.4
```

Alternatively, use GitHub Container Registry:

```
docker run --name prompp -d -p 127.0.0.1:9090:9090 ghcr.io/deckhouse/prompp:0.7.4
```

After launching, Deckhouse Prom++ will be available at http://localhost:9090/.

You can also add your own Prom++ configuration by passing the `--config.file` parameter:

```
docker run --name prompp -v /path/on/host/prometheus.yml:/etc/prometheus.yml -d -p
127.0.0.1:9090:9090 registry.deckhouse.io/prompp/prompp:0.7.4 --config.file=/etc/
prometheus.yml
```

Prometheus Operator

1. Create a file named `prompp.yaml` with the following configuration (other settings may depend on your installation):

   ```
   apiVersion: monitoring.coreos.com/v1
   kind: Prometheus
   metadata:
     name: example-prometheus
     namespace: monitoring
   spec:
     image: registry.deckhouse.io/prompp/prompp:0.7.4  # Replace Prometheus with
   Deckhouse Prom++.
     securityContext:
       fsGroup: 64535
       runAsGroup: 64535
       runAsNonRoot: true
       runAsUser: 64535
     # Additional parameters may be required based on your installation.
   ```

2. Apply the updated resource:

   ```
   kubectl apply -f prompp.yaml
   ```

Migration from Prometheus

Manual WAL conversion

If migration is performed manually, use the `prompptool` utility included in the release.

Converting Prometheus WAL to Deckhouse Prom++

To convert Prometheus WAL to the Deckhouse Prom++ format, run the following command:

```
prompptool walvanilla --working-dir <path to prometheus data dir>
```

Converting Deckhouse Prom++ WAL back to Prometheus

To convert Deckhouse Prom++ WAL to the Prometheus format, run the following command:

```
prompptool walpp --working-dir <path to prometheus data dir>
```

Automatic WAL conversion using Prometheus Operator

Converting Prometheus WAL to Deckhouse Prom++

1. Create a file named `prompp-migration.yaml` with the following configuration (additional parameters may depend on your installation):

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: example-prometheus
  namespace: monitoring
spec:
  ...
  image: registry.deckhouse.io/prompp/prompp:0.7.4
  securityContext:
    fsGroup: 64535
    runAsGroup: 64535
    runAsNonRoot: true
    runAsUser: 64535
  initContainers:
    - name: prompptool
      image: prompp/prompp:<tag>
      command:
        - /bin/prompptool
        - "--working-dir=/prometheus"
        - "walvanilla"
      volumeMounts:
        - name: prometheus-main-db
          mountPath: /prometheus
          subPath: prometheus-db
      resources:
        requests:
          cpu: "100m"
          memory: "128Mi"
  # Additional parameters may be required based on your installation.
```

2. Apply the updated resource:

```
kubectl apply -f prompp-migration.yaml
```

Converting Deckhouse Prom++ WAL back to Prometheus

1. Edit `initContainer` in your `prompp-migration.yaml` file:

```
command:
  - /bin/prompptool
  - "--working-dir=/prometheus"
  - "--verbose"
  - "walpp"
```

2. Apply the changes:

```
kubectl apply -f prompp-migration.yaml
```

Getting started

This is a “Hello, World!"-style guide that demonstrates how to install, configure, and use a simple instance of Deckhouse Prom++. You will download and run Deckhouse Prom++ locally, configure it to collect metrics about itself and a test application, and then work with queries, rules, and graphs to work with the collected time series.

Downloading and running Deckhouse Prom++

Install the latest version of Deckhouse Prom++ for your platform.

Then, configure Deckhouse Prom++ before launching it.

Configuring Deckhouse Prom++ to monitor itself

Deckhouse Prom++ collects metrics from target objects by sequentially scraping their HTTP endpoints with metrics. Since Deckhouse Prom++ exposes data about its own status, it can also monitor its own status. Although a Deckhouse Prom++ server that only collects data about itself is not very useful, it can be a good starting example.

Save the following basic configuration to a file named `prometheus.yml`:

```
global:
  scrape_interval: 15s # By default, target objects are scraped every 15 seconds.
These labels will be attached to all time series and alerts when interacting with
external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'
```

Scrape configuration with a single endpoint

In this case, it is Deckhouse Prom++ itself.

```
scrape_configs:
  # The job name is added to the `job=<job_name>` label for all time series from this
job.
  - job_name: ‘promplusplus’

  # Override the global scrape interval — for this job, it will be 5 seconds.
    scrape_interval: 5s
    static_configs:
      - targets: [‘localhost:9090’]
```

Starting Deckhouse Prom++

To start Deckhouse Prom++ with a new configuration file, navigate to the directory containing the Deckhouse Prom++ binary file and run:

```
# By default, the Prom++ database is stored in ./data (option --storage.tsdb.path).
./prompp --config.file=prometheus.yml
```

Deckhouse Prom++ should start successfully.

Open your browser and go to the Deckhouse Prom++ status page at http://localhost:9090. Give Prom++ a little time to collect data about itself via its own HTTP metrics endpoint.

To ensure Deckhouse Prom++ publishes its metrics, go to http://localhost:9090/metrics.

Using the expression browser

To examine the data that Deckhouse Prom++ has collected about itself, go to the built-in expression browser at http://localhost:9090/graph → “Graph” tab → “Table” view.

As you can see on the page http://localhost:9090/metrics, one of the metrics that Deckhouse Prom++ exports about itself is `prometheus_target_interval_length_seconds` (the actual interval between target object scrapes). Enter the following expression in the query console and click “Execute”:

```
prometheus_target_interval_length_seconds
```

This will return several time series (with the latest stored values), each with the metric name `prometheus_target_interval_length_seconds`, but with different labels. These labels indicate different percentile delays and target grouping intervals.

If you are only interested in the 99th percentile latencies, use the following query:

```
prometheus_target_interval_length_seconds{quantile="0.99"}
```

To count the number of returned time series, use the following query:

```
count(prometheus_target_interval_length_seconds)
```

Using the graph interface

To build a graph based on expressions, go to http://localhost:9090/graph → “Graph” tab.

For example, to build a graph of the rate of new chunks created in the self-scraped Deckhouse Prom++ instance, use the following query:

```
rate(prometheus_tsdb_head_chunks_created_total[1m])
```

Try experimenting with range parameters and other settings.

Reloading configuration without restarting

Deckhouse Prom++ lets you apply a new configuration without restarting by sending a `SIGHUP` signal to the process:

```
kill -s SIGHUP <PID>
```

Correctly terminating Deckhouse Prom++

To correctly terminate Deckhouse Prom++, use the `SIGTERM` signal:

```
kill -s SIGTERM <PID>
```

Configuration

Configuration

Deckhouse Prom++ is configured using command line parameters and a configuration file. Command line parameters set immutable system characteristics (such as storage location, amount of data stored on disk and in memory, etc.), while the configuration file describes everything related to polling jobs and their instances, and also determines which [rule files to load](../ recording_rules/#configuring-rules).

To see a list of all available flags, run `./prompp -h`.

Deckhouse Prom++ can reload its configuration on the fly. If the new configuration is not formed correctly, the changes will not be applied. A configuration reload is triggered by sending a `SIGHUP` signal to the Deckhouse Prom++ process or by sending an HTTP POST request to the `/-/reload` endpoint (when the `--web.enable-lifecycle` flag is enabled). This will also reload all configured rule files.

Configuration file

To explicitly specify which configuration file to load, use the `--config.file` flag.

The file is written in YAML format according to the schema described below. Square brackets indicate optional parameters. For scalar parameters, the default value is used if omitted.

Generalized placeholders are defined as follows:

- `<boolean>`: logical value `true` or `false`

- `<duration>` : duration matching the regular expression `((([0-9]+)y)?(([0-9]+)w)?(([0-9]+)d)?(([0-9]+)h)?( ([0-9]+)m)?(([0-9]+)s)?(([0-9]+)ms)?|0)` , for example `1d` , `1h30m` , `5m` , `10s`
- `<filename>` : valid path in the current working directory
- `<float>` : floating point number
- `<host>` : valid string consisting of a hostname or IP address with an optional port number
- `<int>` : integer
- `<labelname>` : string matching the regular expression `[a-zA-Z_][a-zA-Z0-9_]*` . Any other unsupported characters in the original label must be converted to underscores. For example, the label `app.kubernetes.io/name` must be written as `app_kubernetes_io_name`
- ◦ `<labelvalue>` : Unicode character string
- `<path>` : valid URL path
- `<scheme>` : string accepting values `http` or `https`
- `<secret>` : a regular string containing a secret, such as a password
- `<string>` : a regular string
- `<size>` : size in bytes, for example `512MB` . The unit of measurement is required. Supported units: B, KB, MB, GB, TB, PB, EB
- `<tmpl_string>` : a string that is templated before use.

The remaining placeholders are described separately.

View an example of a correct file.

The global configuration sets parameters that apply to all other configuration contexts. They also serve as default values for other configuration sections.

```
global:
  # How often to poll targets by default.
  [ scrape_interval: <duration> | default = 1m ]

  # The time until the poll request times out.
  # It cannot exceed the poll interval.
  [ scrape_timeout: <duration> | default = 10s ]

  # Protocols to negotiate with the client during polling.
  # Supported values (case-sensitive): PrometheusProto, OpenMetricsText0.0.1,
  # OpenMetricsText1.0.0, PrometheusText0.0.4.
  # The default value is changed to [ PrometheusProto, OpenMetricsText1.0.0,
OpenMetricsText0.0.1, PrometheusText0.0.4 ]
  # when the native_histogram function flag is set.
  [ scrape_protocols: [<string>, ...] | default = [ OpenMetricsText1.0.0,
OpenMetricsText0.0.1, PrometheusText0.0.4 ] ]

  # How often to evaluate rules.
  [ evaluation_interval: <duration> | default = 1m ]

  # Offset the timestamp of the evaluation of rules for this particular group by
  # the specified duration into the past to ensure that
  # the source metrics have been received. Delays in metric availability
  # are more likely when Deckhouse Prom++ is running as a remote recording target,
  # but can also occur during polling anomalies.
  [ rule_query_offset: <duration> | default = 0s ]

  # Labels that are added to any time series or alerts when interacting with
  # external systems (federation, remote storage, Alertmanager).
  # References to environment variables `${var}` or `$var` are replaced according to
  # the values of the current environment variables.
  # References to undefined variables are replaced with an empty string.
  # The `$` symbol can be escaped using `$$`.
  external_labels:
    [ <labelname>: <labelvalue> ... ]

  # File where PromQL queries are written.
  # Reloading the configuration will cause the file to be reopened.
  [ query_log_file: <string> ]

  # File where polling errors are logged.
  # Restarting the configuration will cause the file to be reopened.
  [ scrape_failure_log_file: <string> ]

  # An uncompressed response body exceeding this number of bytes will cause
  # the query to fail. 0 means no limit. Example: 100MB.
  # This is an experimental feature, behavior may change or be removed in the future.
```

```
  [ body_size_limit: <size> | default = 0 ]

  # Limit on the number of samples accepted per poll.
  # If, after label changes, the number of samples exceeds this value,
  # the entire poll will be considered a failure. 0 means no limit.
  [ sample_limit: <int> | default = 0 ]

  # Limit on the number of labels accepted per sample. If, after changing the labels,
  # the number of labels exceeds this value, the entire survey will be considered
unsuccessful. 0 means no limit.
  [ label_limit: <int> | default = 0 ]

  # Limit on the length (in bytes) of each individual label name. If any label name

# in the query is longer than this value after changing the labels, the entire query
will be considered unsuccessful.

# Note that label names are encoded in UTF-8, and characters can take up to 4 bytes. 0
means no limit.
  [ label_name_length_limit: <int> | default = 0 ]

  # Limit on the length (in bytes) of each individual label value. If any label value
  # in the poll is longer than this value after the labels are changed, the entire
poll will be considered a failure.
  # Note that label values are encoded in UTF-8, and characters can take up to 4
bytes. 0 means no limit.
  [ label_value_length_limit: <int> | default = 0 ]

  # Limit on the number of unique targets for each poll configuration
  # that will be accepted. If, after changing labels, the number of targets
  # exceeds this value, Deckhouse Prom++ will mark the targets as failed
  # without polling them. 0 means no limit. This is an experimental
  # feature, behavior may change in the future.
  [ target_limit: <int> | default = 0 ]

  # Limit on the number of targets discarded when changing labels
  # that will be kept in memory for each poll configuration.
  # 0 means no limit.
  [ keep_dropped_targets: <int> | default = 0 ]

  # Specifies the validation scheme for metric names and labels. Either empty,
  # or “utf8” for full UTF-8 support, or “legacy” for letters,
  # numbers, colons, and underscores.
  [ metric_name_validation_scheme <string> | default “utf8” ]

  # Specifies whether to convert all collected classic
  # histograms to native histograms with custom bins.
  [ convert_classic_histograms_to_nhcb <bool> | default = false]
```

```
runtime:
  # Set the GOGC parameter for the Go garbage collector.
  # See https://tip.golang.org/doc/gc-guide#GOGC.
  # Decreasing this number increases CPU usage.
  [ gogc: <int> | default = 75 ]

# Rule files specify a list of templates. Rules and alerts are read
# from all relevant files.
rule_files:
  [ - <filepath_glob> ... ]

# Survey configuration files specify a list of templates. Scrape configurations
# are read from all relevant files and added to the list of scrape configurations.
scrape_config_files:
  [ - <filepath_glob> ... ]

# List of scrape configurations.
scrape_configs:
  [ - <scrape_config> ... ]

# Alerts indicate settings related to Alertmanager.
alerting:
  alert_relabel_configs:
    [ - <relabel_config> ... ]
  alertmanagers:
    [ - <alertmanager_config> ... ]

# Settings related to the remote write feature.
remote_write:
  [ - <remote_write> ... ]

# Settings related to the OTLP receiver feature.
# See https://prometheus.io/docs/guides/opentelemetry/ for best practices.
otlp:
  [ promote_resource_attributes: [<string>, ...] | default = [ ] ]
  # Configures the translation of OTLP metrics when received via the OTLP metrics
endpoint.
  # Available values:
  # - “UnderscoreEscapingWithSuffixes” refers to the common normalization
  #   used by OpenTelemetry in https://github.com/open-telemetry/opentelemetry-
collector-contrib/tree/main/pkg/translator/prometheus

# - “NoUTF8EscapingWithSuffixes” is a mode that relies on UTF-8 support in Deckhouse
Prom++.
  #   It preserves all special characters such as dots, but still adds the necessary
suffixes
  #   for metric names for units and _total, as UnderscoreEscapingWithSuffixes does.
  # - (EXPERIMENTAL) “NoTranslation” is a mode that relies on UTF-8 support in
Deckhouse Prom++.
```

```
  #   It preserves all special characters such as dots and will not add special
suffixes
  #   for units and metric types.
  #
  #   WARNING: The “NoTranslation” setting has significant known risks and limitations
  #   (see https://prometheus.io/docs/practices/naming/ for details):
  #       * Degraded UX when using PromQL in plain YAML (e.g., alerts, rules,
dashboards, autoscaling configuration).
  #       * Series collisions, which at best can lead to OOO errors, at worst to a
silently corrupted
  #         time series. For example, you may find yourself in a situation where you
load a series `foo.bar` with a unit of
  #         `seconds` and a separate series `foo.bar` with a unit of `milliseconds`.

  [ translation_strategy: <string> | default = “UnderscoreEscapingWithSuffixes” ]
  # Enables adding the “service.name”, “service.namespace”, and “service.instance.id”
resource attributes
  # to the “target_info” metric, in addition to converting them to ‘instance’ and
“job” labels.
  [ keep_identifying_resource_attributes: <boolean> | default = false]
  # Configures optional conversion of explicit histograms with OTLP bins to native
histograms with custom bins.
  [ convert_histograms_to_nhcb: <boolean> | default = false]

# Settings related to the remote read feature.
remote_read:
  [ - <remote_read> ... ]

# Storage-related settings that can be reloaded during runtime.
storage:
  [ tsdb: <tsdb> ]
  [ exemplars: <exemplars> ]

# Tracing export configuration.
tracing:
  [ <tracing_config> ]
```

scrape_config

Alerting rules

Derived metrics

Template reference

Template examples

PromQL query language

Basics

Deckhouse Prom++ provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data in real time. The expression result can be represented as a graph, viewed as spreadsheet data in the Deckhouse Prom++ expression browser, or used by external systems via the HTTP API.

Examples

This page is a PromQL basic language reference. For learning, it may be easier to start with a couple of examples.

Expression language data types

In Deckhouse Prom++ expression language, an expression or sub-expression can evaluate to one of four types:

- **Instant vector**: A set of time series containing a single sample for each time series, all sharing the same timestamp.
- **Range vector**: A set of time series containing a range of data points over time for each time series.
- **Scalar**: A simple numeric floating point value.
- **String**: A simple string value; currently unused.

Depending on the use case (for example, when building a graph or displaying the output of an expression), only some of these types are allowed as the result of an expression. For instant, an expression that evaluates to an instant vector is the only type that can be used on a graph.

Literals

String literals

String literals are designated by single quotes (`'`), double quotes (`"`) or backticks (`` ` ``).

PromQL follows the same escaping rules as Go. For string literals in single or double quotes, a backslash begins an escape sequence, which may be followed by `a`, `b`, `f`, `n`, `r`, `t`, `v`, or `\`. Specific characters can be provided using octal (`\nnn`) or hexadecimal (`\xnn`, `\unnnn`, and `\Unnnnnnnn`) notations.

Conversely, escape characters are not parsed in string literals designated by backticks. It is important to note that, unlike Go, Deckhouse Prom++ does not discard newlines inside backticks.

Examples:

```
"this is a string"
'these are unescaped: \n \\ \t'
`these are not unescaped: \n ' " \t`
```

Number literals

Scalar float values can be written as literal integer or floating-point numbers in the following format (whitespace only included for better readability):

```
[-+]?(
      [0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
    | 0[xX][0-9a-fA-F]+
    | [nN][aA][nN]
    | [iI][nN][fF]
)
```

Examples:

```
23
-2.43
3.4e-9
0x8f
-Inf
NaN
```

Time series selectors

Time series selectors are responsible for selecting time series and their corresponding raw or computed values and time labels.

Time series selectors should not be confused with the higher-level concepts of instant and range queries that may execute these selectors. An instant query evaluates a given selector at a single point in time, whereas a range query evaluates the selector at multiple points in time between the start and end timestamps at regular intervals.

Instant vector selectors

Instant vector selectors allow the selection of a set of time series and a single sample value for each at a given timestamp (point in time). In the simplest form, only a metric name is specified, which results in an instant vector containing elements for all time series that have this metric name.

This example selects all time series that have the `http_requests_total` metric name:

```
http_requests_total
```

It is possible to filter these time series further by appending a comma-separated list of label matchers in curly braces (`{}`).

This example selects only those time series with the `http_requests_total` metric name that also have the `job` label set to `Deckhouse Prom++` and their `group` label set to `canary` :

```
http_requests_total{job="Deckhouse Prom++",group="canary"}
```

It is also possible to negatively match a label value, or to match label values against regular expressions. The following label matching operators exist:

- `=` : Select labels that are exactly equal to the provided string.
- `!=` : Select labels that are not equal to the provided string.
- `=~` : Select labels that regex-match the provided string.
- `!~` : Select labels that do not regex-match the provided string.

Regex matches are fully anchored. A match of `env=~"foo"` is treated as `env=~"^foo$"` .

For example, this selects all `http_requests_total` time series for `staging` , `testing` , and `development` environments and HTTP methods other than `GET` .

```
http_requests_total{environment=~"staging|testing|development",method!="GET"}
```

Range vector selectors

Range vector literals work like instant vector literals, except that they select a range of samples back from the current instant. Syntactically, a float literal is appended in square brackets (`[]`) at the end of a vector selector to specify for how many seconds back in time values should be fetched for each resulting range vector element. The range is a enclosed interval, meaning samples with timestamps coinciding with the range boundaries are still included in the selection.

In this example, we select all the values recorded over the last 5 minutes for all time series that have the metric name `http_requests_total` and a `job` label set to `Deckhouse Prom++` :

```
http_requests_total{job="Deckhouse Prom++"}[5m]
```

Time durations

Time durations are specified as a number immediately followed by one of the following units:

- `ms` : Milliseconds
- `s` : Seconds
- `m` : Minutes
- `h` : Hours
- `d` : Days (a day is always assumed to be 24 hours)
- `w` : Weeks (a week is always assumed to be 7 days)
- `y` : Years (a year is always assumed to be 365 days)

> Leap days are ignored when calculating years, and leap seconds are ignored when calculating minutes.

Time durations can be combined by concatenation. Units must be ordered from the longest to the shortest. Each unit may appear only once within a single time duration.

Below are some examples of valid time durations:

```
5h
1h30m
5m
10s
```

Query modifiers

Offset modifier

The `offset` modifier allows changing the time offset for individual instant and range vectors in a query.

For example, the following expression returns the value of `http_requests_total` 5 minutes in the past relative to the current query evaluation time:

```
http_requests_total offset 5m
```

⚠ The `offset` modifier always needs to follow the selector immediately, meaning the following would be correct:

```
sum(http_requests_total{method="GET"} offset 5m) // VALID
```

The following would be incorrect:

```
sum(http_requests_total{method="GET"}) offset 5m // INVALID
```

The same works for range vectors. This returns the 5-minute `rate` that `http_requests_total` had a week ago:

```
rate(http_requests_total[5m] offset 1w)
```

When querying for samples in the past, a negative offset will enable temporal comparisons forward in time:

```
rate(http_requests_total[5m] offset -1w)
```

ℹ This allows a query to look ahead of its evaluation time.

@ modifier

The `@` modifier allows changing the evaluation time for individual instant and range vectors in a query. The time supplied to the `@` modifier is a Unix timestamp and described with a float literal.

For example, the following expression returns the value of `http_requests_total` at `2021-01-04T07:40:00+00:00`:

```
http_requests_total @ 1609746000
```

⚠ The `@` modifier always needs to follow the selector immediately, meaning the following would be correct:

```
sum(http_requests_total{method="GET"} @ 1609746000) // VALID
```

The following would be incorrect:

```
sum(http_requests_total{method="GET"}) @ 1609746000 // INVALID
```

The same works for range vectors. This returns the 5-minute `rate` that `http_requests_total` had at `2021-01-04T07:40:00+00:00`:

```
rate(http_requests_total[5m] @ 1609746000)
```

The `@` modifier supports all representations of numeric literals described above. It works with the `offset` modifier where the offset is applied relative to the `@` modifier time. The results are the same irrespective of the order of the modifiers.

For example, these two queries will produce the same result:

```
# offset after @
http_requests_total @ 1609746000 offset 5m
# offset before @
http_requests_total offset 5m @ 1609746000
```

Additionally, `start()` and `end()` can also be used as values for the `@` modifier as special values.

For a range query, they resolve to the start and end of the range query respectively and remain the same for all steps.

For an instant query, `start()` and `end()` both resolve to the evaluation time.

Examples:

```
http_requests_total @ start()
rate(http_requests_total[5m] @ end())
```


The `@` modifier allows a query to look ahead of its evaluation time.

Subqueries

A subquery allows you to run an instant query for a given range and resolution. The result of a subquery is a range vector.

Syntax:

```
<instant_query> '[' <range> ':' [<resolution>] ']' [ @ <float_literal> ] [ offset
<duration> ]
```

where:

- is optional. Default is the global evaluation interval.

Operators

Deckhouse Prom++ supports many binary and aggregation operators. These are described in detail in PromQL operators.

Functions

Deckhouse Prom++ supports several functions to operate on data. These are described in detail in the expression language functions page.

Comments

PromQL supports inline comments that start with `#`. Example:

```
# This is a comment.
   metric{label="value"} # This is another comment.
```

Gotchas

Staleness

The timestamps at which to sample data, during a query, are selected independently of the actual present time series data. This is mainly to support cases like aggregation (`sum`, `avg`, and so on), where multiple aggregated time series do not precisely align in time. Because of their independence, Deckhouse Prom++ needs to assign a value at those timestamps for each relevant time series. It does so by taking the newest sample that is less than the lookback period ago. The lookback period is 5 minutes by default.

If a target scrape or rule evaluation no longer returns a sample for a time series that was previously present, this time series will be marked as stale. If a target is removed, the previously retrieved time series will be marked as stale soon after removal.

If a query is evaluated at a sampling timestamp after a time series is marked as stale, then no value is returned for that time series. If new samples are subsequently ingested for that time series, they will be returned as expected.

A time series will go stale when it is no longer exported, or the target no longer exists. Such time series will disappear from graphs at the times of their latest collected sample, and they will not be returned in queries after they are marked stale.

Some exporters, which put their own timestamps on samples, get a different behaviour: series that stop being exported take the last value for 5 minutes (by default) before disappearing. The `track_timestamps_staleness` setting can change this.

Avoiding slow queries and overloads

If a query needs to operate on a substantial amount of data, graphing it might time out or overload the server or browser. Thus, when constructing queries over unknown data, always start building the query

in the tabular view of Deckhouse Prom++ expression browser until the result set seems reasonable (hundreds, not thousands, of time series at most). Only when you have filtered or aggregated your data sufficiently, switch to graph mode. If the expression still takes too long to graph ad-hoc, pre-record it via a recording rule.

This is especially relevant for Deckhouse Prom++ query language, where a bare metric name selector like `api_http_requests_total` could expand to thousands of time series with different labels. Also, keep in mind that expressions that aggregate over many time series will generate load on the server even if the output is only a small number of time series. This is similar to how it would be slow to sum all values of a column in a relational database, even if the output value is only a single number.

PromQL operators

PromQL functions

Examples

Remote read API

Deckhouse Prom++ provides an API for remote read, accessible via the endpoint `/api/v1/read`. This interface expects compression using the Snappy algorithm. For more information about the API, visit the Prom++ repository.

> **Note:** This API is currently not considered stable and may change even between minor versions of Deckhouse Prom++.

Samples

This endpoint returns a message containing a list of raw samples matching the requested query.

Streamed Chunks

Streamed chunks use an XOR algorithm inspired by Gorilla compression to encode chunks. However, it provides millisecond resolution instead of second resolution.

HTTP API

Federation

Federation allows a Deckhouse Prom++ server to scrape selected time series from another Deckhouse Prom++ server.

Use cases

There are different use cases for federation. Commonly, it is used to either achieve scalable Deckhouse Prom++ monitoring setups or to pull related metrics from one service into another.

Hierarchical federation

Hierarchical federation allows Deckhouse Prom++ to scale to environments with tens of data centers and millions of nodes. In this use case, the federation topology resembles a tree, with higher-level Deckhouse Prom++ servers collecting aggregated time series data from a large number of subordinated servers.

For example, a setup might consist of many Deckhouse Prom++ servers in each datacenter that collect data in high detail (instance-level), and a set of global Deckhouse Prom++ servers, which collect and store only aggregated data (job-level) from those local servers. This provides an aggregate global view and detailed local views.

Cross-service federation

In cross-service federation, a Deckhouse Prom++ server of one service is configured to scrape selected data from another service's Deckhouse Prom++ server to enable alerting and queries against both datasets within a single server.

For example, a cluster scheduler running multiple services might expose resource usage information (like memory and CPU usage) about service instances running in the cluster. On the other hand, a service running in that cluster will only expose application-specific service metrics. Often, these two sets of metrics are scraped by separate Deckhouse Prom++ servers. Using federation, the Deckhouse Prom++ server containing service-level metrics may pull in the cluster resource usage metrics about its specific service from the cluster Deckhouse Prom++, so that both sets of metrics can be used within that server.

Configuring federation

On any Deckhouse Prom++ server, the `/federate` endpoint allows retrieving the current value for a selected set of time series in that server. At least one `match[]` URL parameter must be specified to select the series to expose. Each `match[]` argument needs to specify an instant vector selector, such as `up` or `{job="api-server"}`. If multiple `match[]` parameters are provided, the union of all matched series is selected.

To federate metrics from one server to another, configure your destination Deckhouse Prom++ server to scrape from the `/federate` endpoint of a source server, while also enabling the `honor_labels` scrape option (to prevent overwriting any labels exposed by the source server) and passing in the desired `match[]` parameters. For example, the following `scrape_configs` configuration scrapes any series with the label `job="PromPP"` or a metric name starting with `job:` from the Deckhouse Prom++ servers at `source-PromPP-{1,2,3}:9090` into the collecting Deckhouse Prom++ server:

```
scrape_configs:
  - job_name: 'federate'
    scrape_interval: 15s

    honor_labels: true
    metrics_path: '/federate'

    params:
      'match[]':
        - '{job="PromPP"}'
        - '{__name__=~"job:.*"}'

    static_configs:
      - targets:
        - 'source-PromPP-1:9090'
        - 'source-PromPP-2:9090'
        - 'source-PromPP-3:9090'
```

HTTP Service Discovery (HTTP SD)

Deckhouse Prom++ provides a generic HTTP Service Discovery (HTTP SD) mechanism that enables it to discover targets over an HTTP endpoint.

HTTP SD is complementary to the supported service discovery mechanisms, and is an alternative to File-based SD.

Comparison between File-based SD and HTTP SD

Below is a table comparing two generic Service Discovery implementations.

Parameter	File-based SD	HTTP SD
Event-based	Yes, via inotify	No
Update frequency	Instant, thanks to inotify	Following `refresh_interval`
Format	YAML or JSON	JSON
Transport	Local file	HTTP/HTTPS
Безопасность	File permissions	TLS, Basic auth, Authorization header, OAuth2

Requirements to HTTP SD endpoints

If you implement an HTTP SD endpoint, here are a few requirements you should be aware of.

The response is consumed as is, unmodified. On each refresh interval (default: 1 minute), Deckhouse Prom++ will perform a GET request to the HTTP SD endpoint. The GET request contains a `X-Deckhouse Prom++-Refresh-Interval-Seconds` HTTP header with the refresh interval.

The SD endpoint must answer with an HTTP 200 response, with the HTTP Header Content-Type: application/json. The answer must be UTF-8 formatted. If no targets should be transmitted, HTTP 200 must also be emitted, with an empty list `[]`. Target lists are unordered.

Deckhouse Prom++ caches target lists. If an error occurs while fetching an updated targets list, Deckhouse Prom++ keeps using the current targets list. The targets list is not saved across restart. The `prompp_sd_http_failures_total` counter metric tracks the number of refresh failures.

The whole list of targets must be returned on every scrape. There is no support for incremental updates. A Deckhouse Prom++ instance does not send its hostname and it is not possible for a SD endpoint to know if the SD request is the first one after a restart or not.

The URL to the HTTP SD is not considered secret. The authentication and any API keys should be passed with the appropriate authentication mechanisms. Deckhouse Prom++ supports TLS authentication, basic authentication, OAuth2, and authorization headers.

HTTP SD format

```
[
  {
    "targets": [ "<host>", ... ],
    "labels": {
      "<labelname>": "<labelvalue>", ...
    }
  },
  ...
]
```

Examples:

```
[
    {
        "targets": ["10.0.10.2:9100", "10.0.10.3:9100", "10.0.10.4:9100", "10.0.10.5:9
100"],
        "labels": {
            "__meta_datacenter": "london",
            "__meta_prompp_job": "node"
        }
    },
    {
        "targets": ["10.0.40.2:9100", "10.0.40.3:9100"],
        "labels": {
            "__meta_datacenter": "london",
            "__meta_prompp_job": "alertmanager"
        }
    },
    {
        "targets": ["10.0.40.2:9093", "10.0.40.3:9093"],
        "labels": {
            "__meta_datacenter": "newyork",
            "__meta_prompp_job": "alertmanager"
        }
    }
]
```

Command line

Below is the list of available Deckhouse Prom++ command line flags, along with descriptions.

Flag	Description	Default value
`-h` , `--help`	Show context-sensitive help (also via `--help-long` and `--help-man`)	
`--version`	Show application version	
`--config.file`	Deckhouse Prom++ configuration file path	`prometheus.yml`
`--web.listen-address`	Address to listen on for UI, API, and telemetry.	`0.0.0.0:9090`
`--auto-gomemlimit.ratio`	Ratio of reserved GOMEMLIMIT memory to the maximum system memory	`0.9`
`--web.config.file`	(**Experimental**) Path to configuration file that can enable TLS or authentication	
`--web.read-timeout`	Maximum duration before request reading times out	`5m`
`--web.max-connections`	Maximum number of simultaneous connections	`512`
`--web.external-url`	External URL for accessing Deckhouse Prom++ (via a reverse proxy)	
`--web.route-prefix`	Prefix for the internal routes of the web interface	
`--web.user-assets`	Path to static assets (`/user`)	
`--web.enable-lifecycle`	Enable shutdown and reload via HTTP requests	`false`
`--web.enable-admin-api`	Enable API for admin control actions	`false`
`--web.enable-remote-write-receiver`	Enable API accepting remote write requests	`false`
`--web.console.templates`		`consoles`

Flag	Description	Default value
	Path to the console templates (`/consoles`)	
`--web.console.libraries`	Path to the console libraries	`console_libraries`
`--web.page-title`	Page title	`Deckhouse Prom++ Time Series Collection and Processing Server`
`--web.cors.origin`	Regex for CORS origin	
`--storage.tsdb.path`	Path to metrics data (server mode).	`data/`
`--storage.tsdb.retention`	(**Deprecated**) Sample retention period	
`--storage.tsdb.retention.time`	Sample retention period (overrides `retention`)	
`--storage.tsdb.retention.size`	Maximum storage volume (for example, `512MB`)	
`--storage.tsdb.no-lockfile`	Do not create lockfile	`false`
`--storage.tsdb.head-chunks-write-queue-size`	Size of the queue through which head chunks are written	`0`
`--storage.agent.path`	Path to data (agent mode)	`data-agent/`
`--storage.agent.wal-compression`	Compress the agent WAL	`true`
`--storage.agent.retention.min-time`	Minimum sample retention time (agent mode)	
`--storage.agent.retention.max-time`	Maximum sample retention time (agent mode)	
`--storage.agent.no-lockfile`	Do not create lockfile (agent mode)	`false`
`--storage.remote.flush-deadline`	Data flush timeout after shutdown	`1m`

Flag	Description	Default value
`--storage.remote.read-sample-limit`	Sample limit for remote read	`5e7`
`--storage.remote.read-concurrent-limit`	Limit for concurrent remote read	`10`
`--storage.remote.read-max-bytes-in-frame`	Maximum frame size for remote read	`1048576`
`--rules.alert.for-outage-tolerance`	Maximum outage period for restoring alerts	`1h`
`--rules.alert.for-grace-period`	Minimum alert interval	`10m`
`--rules.alert.resend-delay`	Delay before resending an alert	`1m`
`--rules.max-concurrent-evals`	Concurrency limit for rules	`4`
`--alertmanager.notification-queue-capacity`	Notification queue capacity	`10000`
`--query.lookback-delta`	Maximum lookback for queries	`5m`
`--query.timeout`	Query execution timeout	`2m`
`--query.max-concurrency`	Maximum number of concurrent queries	`20`
`--query.max-samples`	Maximum number of samples per single query	`50000000`
`--enable-feature`	Enable experimental features	
`--log.level`	Log verbosity (`debug` , `info` , `warn` , `error`)	`info`
`--log.format`	Log format (`logfmt` , `json`)	`logfmt`

Management API

Deckhouse Prom++ provides a set of management APIs to facilitate automation and integration.

Health check

```
GET /-/healthy
HEAD /-/healthy
```

This endpoint always returns the `200` status and should be used to check Deckhouse Prom++ health.

Readiness check

```
GET /-/ready
HEAD /-/ready
```

This endpoint returns the `200` status when Deckhouse Prom++ is ready to serve traffic (meaning, respond to queries).

Reload

```
PUT  /-/reload
POST /-/reload
```

This endpoint triggers a reload of the Deckhouse Prom++ configuration and rule files. It's disabled by default and can be enabled via the `--web.enable-lifecycle` flag.

Alternatively, a configuration reload can be triggered by sending a `SIGHUP` to the Deckhouse Prom++ process.

Quit

```
PUT  /-/quit
POST /-/quit
```

This endpoint triggers a graceful shutdown of Deckhouse Prom++. It's disabled by default and can be enabled via the `--web.enable-lifecycle` flag.

Alternatively, a graceful shutdown can be triggered by sending a `SIGTERM` to the Deckhouse Prom++ process.